Introduction to Programming

Week 9

Magnus Madsen

Week 9: Outline

- Object-Oriented Programming
  - Classes and Objects
  - Constructors and Methods
  - Access Modifiers and Encapsulation
  - Static Members
  - References and Aliasing
  - Enums
- Code Style
- Live Programming

Quote of the Week

"Object-oriented programming is a method of implementation in which programs are organized as cooperative collections of objects, each of which represents an instance of some class, and whose classes are all members of a hierarchy of classes united via inheritance relationships."

— Grady Booch

Epigram of the Week

"Programmers must avoid leaving false clues that obscure the meaning of code."

— Robert C. Martin

Object-Oriented Programming

We now introduce **Object-Oriented Programming** (OOP)

In OOP, **classes** define data types, and **objects** are their instances.

Informally, classes are **blueprints** and objects are what we build from them.

Why Object-Oriented Programming?

Building large software requires **abstraction**. Abstraction means:

- Reveal only **relevant** information.
- Hide **irrelevant** implementation details.

OOP achieves this by bundling data and operations into **classes**.

Result: We can use data types without knowing their internal implementation!

Objects = Identity + State + Behavior

Let us break it down:

- **Identity**: Every object is unique and distinct from others.
- **State**: Every object stores its own data (its *fields*).
- **Behavior**: Every object has a collection of operations it can perform (its *methods*).

Classes and Objects

Introduction to Classes

In Java, we define a class using the `class` keyword.

A class contains:

Fields: Variables that store the object's state (its data).

Constructors: Special methods that are used to create new instances of the class.

Methods: Functions that define the object's behavior (its operations).

Together, these components make up the class definition.

Example: Declaring a Class

We can declare a class to model a bank account:

```java
public class BankAccount {
    // Fields
    private final String name;
    private double balance;
    private double rate;

    // Constructor
    public BankAccount(String name, double balance) {
        this.name = name;
        this.balance = balance;
        this.rate = 0.05;
    }
}
```

Remark: We use `this` to refer to the fields and methods of the class.

Note: The name of the constructor must match the name of the class.

Example: Creating an Object

We can create a “fresh” `BankAccount` object using the `new` keyword:

```
BankAccount checking = new BankAccount("Magnus' Checking", 42.0);
```

Example: Creating an Object

We can create a “fresh” `BankAccount` object using the `new` keyword:

```
BankAccount checking = new BankAccount("Magnus' Checking", 42.0);
```

We can create more accounts as needed:

```
BankAccount savings = new BankAccount("Magnus' Savings", 500.0);
```

Note: We have *two* objects. Each object has its own name, balance, and rate.

Constructor Overloading (1/2)

A class can have **multiple constructors** using **overloading**:

```
public BankAccount(String name) {
    this.name = name;
    this.balance = 0;
    this.rate = 0.05;
}

public BankAccount(String name, double balance) {
    this.name = name;
    this.balance = balance;
    this.rate = 0.05;
}
```

Remark: A constructor should ensure that all fields of an object are initialized, though it does not necessarily need to receive all of them as parameters.

Recall, we learned about *method overloading* in Week 6.

Constructor overloading follows the same principle:

- We can overload based on **the number of formal parameters** (arity).
- We can overload based on **the type of the formal parameters**.

Upshot: We can allow an object of a class to be constructed in multiple ways.

Every Java class must have at least one constructor.

- If we fail to provide one, Java automatically inserts a default empty constructor.

Instance Methods (1/2)

We can define **instance methods** to access or modify fields:

```
public double getBalance() {
    return this.balance;
}

public void setRate(double newRate) {
    this.rate = newRate;
}
```

We call such methods *getters* and *setters* (more on that later).

Instance Methods (2/2)

We use **instance methods** to define the behavior of an object:

```java
public void depositBalance(double amount) {
    this.balance = this.balance + amount;
}

public void transferTo(BankAccount dst, double amount) {
    dst.depositBalance(amount);
    this.balance = this.balance - amount;
}
```

Using Objects

An instance of a class is called an **object**.

- We create an instance using the `new` operator.
- Every object has its own identity and its own state.

```java
public class Main {
    public static void main(String[] args) {
        BankAccount account1 = new BankAccount("Checking");
        BankAccount account2 = new BankAccount("Checking");
        BankAccount account3 = new BankAccount("Savings", 100.0);
    }
}
```

Here we have *three* independent objects. Two of them happen to just have the same name.

We invoke (call) a method on an object by writing: `object.method(...)`.

For example, we can write:

```
BankAccount checking = new BankAccount("Checking", 100.0);
BankAccount savings = new BankAccount("Savings", 200.0);
checking.deposit(500);
savings.deposit(500);
```

Example: Invoking Instance Methods (2/2)

We can also write:

```java
BankAccount checking = new BankAccount("Checking", 100.0);
BankAccount savings = new BankAccount("Savings", 200.0);
checking.transferTo(savings, 100.0);
```

Here we call the instance method `transferTo` on the `checking` object passing the `savings` object as an argument.

Mutable vs. Immutable Objects

We distinguish between two kinds of objects:

- A **mutable object** can have its state changed after creation.
- A **immutable object** cannot have its state changed after creation.

Examples:

- `BankAccount` is a mutable object: we can call methods like `deposit()` to change its balance (i.e. internal state).
- `String` is an immutable object: methods like `trim()` and `toUpperCase()` return a *new* String rather than modifying the original.

Method Chaining

In Java, Strings are (immutable) objects with many useful instance methods.

For example, we can write:

```
String s1 = "    hello world!    ";
String s2 = s1.trim();
String s3 = s2.toUpperCase();
StdOut.println(s3) // Prints: "HELLO WORLD!"
```

Using **method chaining** we can also write:

```
String s = "    hello world!    ".trim().toUpperCase();
StdOut.println(s) // Prints: "HELLO WORLD!"
```

What is `final`?

We can use the `final` keyword to declare that the value of a field cannot be changed after it has been initialized.

For example, we can enforce that the name of a `BankAccount` cannot change:

```
public class BankAccount {
    // Fields
    private final String name;
    private double balance;
    private double rate;

    // ...
}
```

Note: The `final` keyword is also used in some other contexts that we shall not concern ourselves with here.

What is `this`?

The `this` keyword refers to the **current object**,

- i.e., the object whose method or constructor is currently being executed.

Common uses of `this`:

- Refer to the object's own fields and methods.
- Pass the current object as an argument to another method.
- Distinguish between fields and parameters with the same name (`this.name = name`).

Which fields should be `final`?

Q:

```
class Book {
    private String isbn;
    private String title;
    private String author;
    private int publicationYear;
    private int pageCount;
    private double rating;
    private int copiesSold;
}
```

Q:

Should these be mutable or immutable objects?

- A shopping cart
- A postal address
- A calendar date
- A bank transaction
- A playlist
- A stopwatch

Multiple Objects

A key feature of object-oriented programming is that objects **collaborate** to accomplish their tasks.

Key concepts:

- Objects can call methods on other objects.
- Objects can contain references to other objects as fields.
- This enables **delegation**: one object can delegate work to another object.

Example: A `Customer` class (1/2)

We can define a `Customer` class which uses the `BankAccount` class:

```java
public class Customer {
    private BankAccount checking;
    private BankAccount savings;
    private BankAccount[] otherBankAccounts = new BankAccount[100];

    public Customer(String name) {
        this.checking = new BankAccount(name + "checking");
        this.savings = new BankAccount(name + "savings");
    }
}
```

Example: A `Customer` class (2/2)

We can define methods on `Customer` that call methods on `BankAccount`:

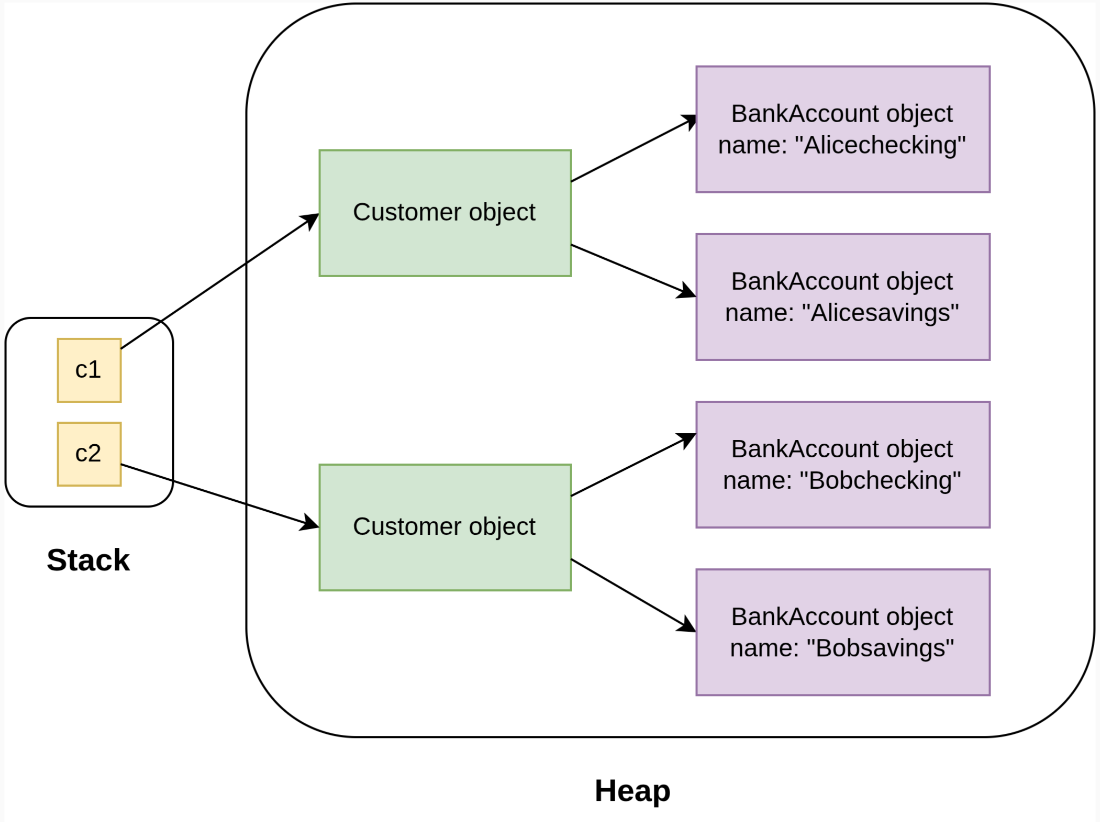
```java
public class Customer {
    // ... as on previous slide ...
    public double getTotal() {
        double total = checking.getBalance() + savings.getBalance();
        for (int i = 0; i < otherBankAccounts.length; i++) {
            total += otherBankAccounts[i].getBalance();
        }
        return total;
    }

    public void transferTo(Customer c, double amount) {
        this.savings.transferTo(c.savings, amount);
    }
}
```

Using `Customer`

We can now use `Customer` in our `main` method:

```
public static void main(String[] args) {
    Customer c1 = new Customer("Alice");
    Customer c2 = new Customer("Bob");
    c2.transfer(c1, 100.0);
}
```



Objects = Identity + State + Behavior

For the `Customer` class we have:

Identity: Each `Customer` object is distinct, even with the same name.

State: Each `Customer` has its own `checking`, `savings`, and `otherBankAccounts` fields.

Behavior: Each `Customer` can perform operations like `getTotal()` and `transferTo()`.

Static vs. Instance Methods

Static vs. Instance Methods

A **static method** belongs to the class itself and is not associated with any particular object.

An **instance method** is called on a specific object and can access that object's state.

Static method:

```
int result = Math.max(5, 10);
```

- Called on class `Math`
- No access to any object's state

Instance method:

```
BankAccount a = new BankAccount("...");
a.deposit(100.0);
```

- Called on object `a`
- Has access to `a`'s state

Static vs. Instance Fields

A **static field** belongs to the class itself. There is only one copy shared by all objects.

An **instance field** belongs to each individual object. Every object has its own copy.

Static field:

```
class BankAccount {
    private static String bankAddress;
}
```

- All `BankAccount` objects share one `bankAddress`
- Changing it affects all objects

Instance field:

```
class BankAccount {
    private double balance;
}
```

- Each `BankAccount` object has its own `balance`
- Different objects have different values

Recall the memoized Fibonacci function.

We can implement it using a class with a static field and static method:

```java
class Fib {
    private static int[] memo = new int[100];

    static int fibMemo(int n) {
        if (n <= 1) return n;
        if (memo[n] != 0) return memo[n];
        memo[n] = fibMemo(n - 1) + fibMemo(n - 2);
        return memo[n];
    }
}
```

Q:

Which fields should be `static`? `final`?

```java
class Employee {
    private String employeeId;
    private String name;
    private double salary;
    private String companyName;
    private int totalEmployees;
}
```

Q:

What does this print?

```java
class BankAccount {
    private static int nextId = 1;
    private int id;
    private double balance;
    public BankAccount(double balance) {
        this.id = nextId++;
        this.balance = balance;
    }
    public int getId() {
        return this.id;
    }
}

public static void main(String[] args) {
    BankAccount a1 = new BankAccount(100.0);
    BankAccount a2 = new BankAccount(200.0);
    System.out.println(a2.getId());
}
```

Access Modifiers

What are Access Modifiers?

Access modifiers control who can access the fields and methods of a class.

The two most common access modifiers are:

- `public` - can be accessed from anywhere
- `private` - can only be accessed within the class itself

Why do we need them?

- Enforce **encapsulation**: hide internal implementation details
- Prevent misuse of fields and methods
- Allow us to change internal implementation without breaking external code

Overview: Access Modifiers

Java has four access modifiers that control visibility in different contexts:

Context	Private	Default	Protected	Public
Same class	✅	✅	✅	✅
Same package subclass	❌	✅	✅	✅
Same package non-subclass	❌	✅	✅	✅
Different package subclass	❌	❌	✅	✅
Different package non-subclass	❌	❌	❌	✅

Remark: A package is pretty much like a directory.

Example: Access Modifier

Recall that the `balance` field in `BankAccount` is `private`.

If we try to access it from outside the `BankAccount` class we get a compile error:

```java
public class Main {
    public static void main(String[] args) {
        BankAccount account = new BankAccount("Checking");
        account.balance = 1000.0;  // ERROR: balance has private access
    }
}
```

To access or modify `balance`, we must use `public` methods like `getBalance()` or `deposit()`.

Getters and Setters

What are Getters and Setters?

Getters and **setters** are special methods that provide **controlled access** to private fields.

- A **getter** is a method that *returns* the value of a private field.
- A **setter** is a method that *modifies* the value of a private field.

Enforces **encapsulation** while providing (limited) access to an object's state.

Example: Getters and Setters

```java
class Person {
    private String ssn;
    private int age;
    public Person(String ssn, int age) {
        this.ssn = ssn;
        this.age = age;
    }
    public String getSsn() {
        return this.ssn;
    }
    public int getAge() {
        return this.age;
    }
    public void setAge(int newAge) {
        if (newAge > 0) {
            this.age = newAge;
        }
    }
}
```

Key Points:

- The `ssn` field has a getter but no setter, hence cannot be changed once assigned.
- The `age` field has both a getter and setter, where the setter ensures the value is non-negative.
- We use getters and setters to enforce:
  - **Encapsulation**
  - **Class invariants**

Q:

What is wrong here?

```
class Rectangle {
    private int width;
    private int height;

    public Rectangle(int width, int height) {
        width = width;
        height = height;
    }
}
```

Null

What is `null`?

`null` is a **special value** that indicates a reference does not point to an object.

- Uninitialized reference variables default to `null`.
- We can explicitly set a field to `null` to indicate “no value”.
- We can check if a reference is `null` using `==`.
- **Calling a method or accessing a field on `null` causes a `NullPointerException`.**

Example: Using `null`

We can use `null` to represent optional data, such as a middle name that may not exist:

```java
class Person {
    private String firstName;
    private String middleName;
    private String lastName;

    public Person(String firstName, String lastName) {
        this.firstName = firstName;
        this.middleName = null;  // No middle name
        this.lastName = lastName;
    }

    public boolean hasMiddleName() {
        return this.middleName != null;
    }
}
```

The Dreaded `NullPointerException`

If you try to call a method of a `null` object, you get a `NullPointerException`!

```
Person p = new Person("George", "Martin");
if (p.getMiddleName().contains("RR")) {  // Crash!
    System.out.println("Winter is coming! (... eventually)");
}
```

The Dreaded `NullPointerException`

If you try to call a method of a `null` object, you get a `NullPointerException`!

```java
Person p = new Person("George", "Martin");
if (p.getMiddleName().contains("RR")) {  // Crash!
    System.out.println("Winter is coming! (... eventually)");
}
```

```
Exception in thread "main" java.lang.NullPointerException: Cannot invoke
"String.contains()" because the return value of "Person.getMiddleName()" is
null at Main.main(Main.java:3)
```

Sir Tony Hoare invented null while designing ALGOL. He later said:

"I call it my billion-dollar mistake. It was the invention of the null reference in 1965. [...] This has led to innumerable errors, vulnerabilities, and system crashes, which have probably caused a billion dollars of pain and damage in the last forty years."

Q:

What can you say about this program?

```
Person p = new Person("Bob", "Johnson");
String fullName = p.firstName + " " + p.middleName + " " + p.lastName;
System.out.println(fullName.trim());
```

Color Class

Color ADT

Color is a sensation in the eye from electromagnetic radiation.

An ADT allows us to write Java programs that manipulate color.

Values

		examples							
R (8 bits)	red intensity	255	0	0	0	255	0	119	105
G (8 bits)	green intensity	0	255	0	0	255	64	33	105
B (8 bits)	blue intensity	0	0	255	0	255	128	27	105
color									

API (operations)

`public class Color`	
`Color(int r, int g, int b)`	
`int getRed()`	*red intensity*
`int getGreen()`	*green intensity*
`int getBlue()`	*blue intensity*
`Color brighter()`	*brighter version of this color*
`Color darker()`	*darker version of this color*
`String toString()`	*string representation of this color*
`boolean equals(Color c)`	*is this color the same as c's ?*

Computing with color: monochrome luminance

Def. The *monochrome luminance* of a color quantifies its effective brightness.

NTSC standard formula for luminance: 0.299r + 0.587g + 0.114b.

```
import java.awt.Color;
public class Luminance
{
   public static double lum(Color c)
   {
      int r = c.getRed();
      int g = c.getGreen();
      int b = c.getBlue();
      return .299*r + .587*g + .114*b;
   }
   public static void main(String[] args)
   {
      int r = Integer.parseInt(args[0]);
      int g = Integer.parseInt(args[1]);
      int b = Integer.parseInt(args[2]);
      Color c = new Color(r, g, b);
      StdOut.println(Math.round(lum(c)));
   }
}
```

```
% java Luminance 0 64 128
52
```

	examples							
red intensity	255	0	0	0	255	0	119	105
green intensity	0	255	0	0	255	64	33	105
blue intensity	0	0	255	0	255	128	27	105
color								
luminance	76	150	29	0	255	52	58	105

Applications (next)

- Choose colors for displayed text.
- Convert colors to grayscale.

Computing with color: grayscale

Goal. Convert colors to grayscale values.

Fact. When all three R, G, and B values are the same, resulting color is on grayscale from 0 (black) to 255 (white).

Q. What value for a given color?

A. Its luminance!

```
public static Color toGray(Color c)
{
   int y = (int) Math.round(lum(c));
   Color gray = new Color(y, y, y);
   return gray;
}
```

method for Luminance library

	examples							
red intensity	255	0	0	0	255	0	119	105
green intensity	0	255	0	0	255	64	33	105
blue intensity	0	0	255	0	255	128	27	105
color								
luminance	76	150	29	0	255	52	58	105
grayscale								

Picture Class

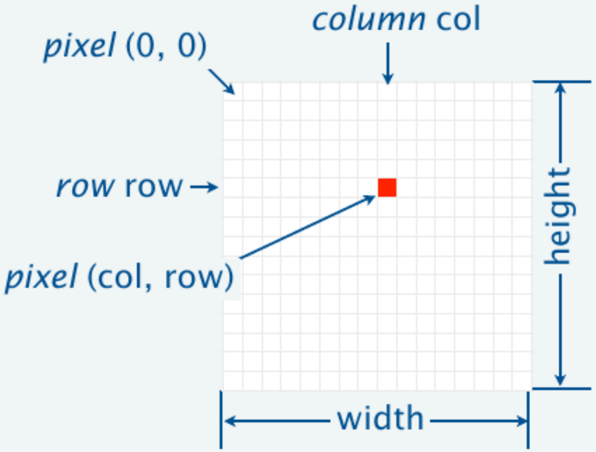
Picture ADT

A Picture is a 2D array of pixels.

defined in terms of its ADT values (typical)

An ADT allows us to write Java programs that manipulate pictures.

Values (2D arrays of Colors)



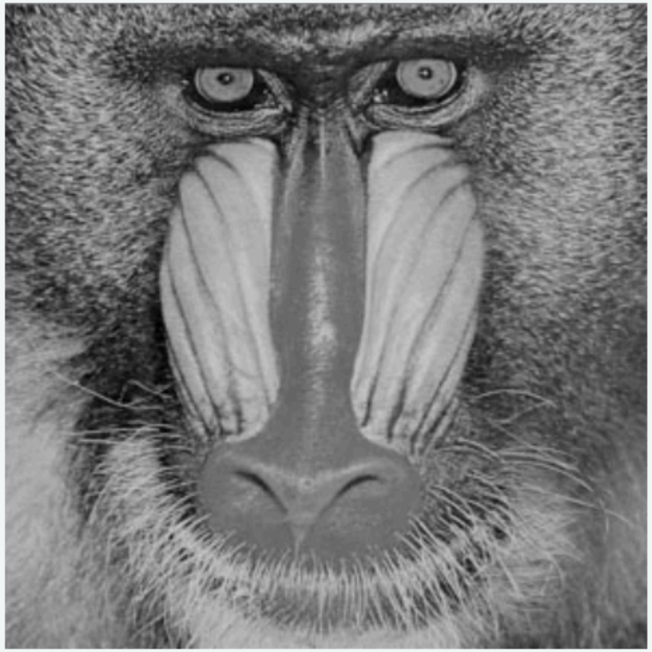
API (operations)

public class Picture	
`Picture(String filename)`	*create a picture from a file*
`Picture(int w, int h)`	*create a blank w-by-h picture*
`int width()`	*width of the picture*
`int height()`	*height of the picture*
`Color get(int col, int row)`	*the color of pixel (col, row)*
`void set(int col, int row, Color c)`	*set the color of pixel (col, row) to c*
`void show()`	*display the image in a window*
`void save(String filename)`	*save the picture to a file*

Picture client example: Grayscale filter

Goal. Write a Java program to convert an image to grayscale.

Source: `mandrill.jpg`



```
% java Grayscale mandrill.jpg
```

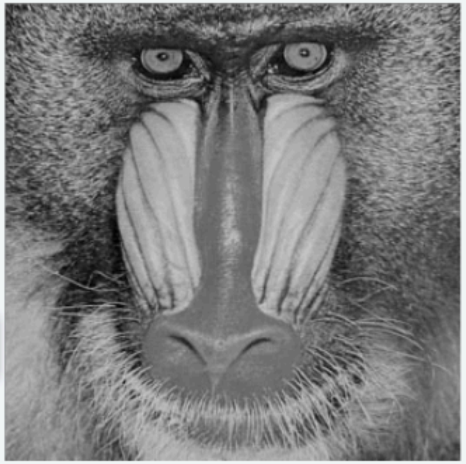
Picture client example: Grayscale filter

```
import java.awt.Color;
public class Grayscale
{
   public static void main(String[] args)
   {
      Picture pic = new Picture(args[0]);
      for (int col = 0; col < pic.width(); col++)
         for (int row = 0; row < pic.height(); row++)
         {
            Color color = pic.get(col, row);
            Color gray  = Luminance.toGray(color);
            pic.set(col, row, gray);
         }
      pic.show();
   }
}
```

create a new picture

fill in each pixel

```
% java Grayscale mandrill.jpg
```



Pop quiz 1b on image processing

Q. What is the effect of the following code (not-so-easy question)?

```
Picture pic = new Picture(args[0]);
for (int col = 0; col < pic.width(); col++)
   for (int row = 0; row < pic.height(); row++)
      pic.set(col, pic.height()-row-1, pic.get(col, row));
pic.show();
```

Pop quiz 1b on image processing

Q. What is the effect of the following code (not-so-easy question)?

```
Picture pic = new Picture(args[0]);
for (int col = 0; col < pic.width(); col++)
   for (int row = 0; row < pic.height(); row++)
      pic.set(col, pic.height()-row-1, pic.get(col, row));
pic.show();
```

A. Tries to turn image upside down, but fails.
An instructive bug!.

Aliasing

What is Aliasing?

Aliasing occurs when two or more references point to the same object.

Key points:

- Changes made through one reference affect all other references.
- This happens because references store memory addresses, not copies of objects.
- Aliasing applies to arrays, objects, and all reference types.

Aliasing is neither good nor bad; it is just a feature.

Example: Aliasing

We create three variables but only two objects:

```
BankAccount checking = new BankAccount("Checking", 100.0);
BankAccount savings = new BankAccount("Savings", 200.0);
BankAccount account = checking;  // Alias! Points to same object as checking
```

Now `checking` and `account` refer to the same object:

```
account.deposit(50.0);
System.out.println(checking.getBalance());  // Prints: 150.0
```

What in the blazes does this do?:

```
checking.transferTo(account, 75.0);
System.out.println(checking.getBalance());  // Prints: ???
```

Comparing References (1/2)

The `==` operator compares **references**, not the **contents** of objects.

For primitive types:

- `==` compares values directly.

For reference types:

- `==` checks if two references point to the **same object in memory**.
- It does **not** compare the contents / fields of the objects.

Comparing References (2/2)

Here is a small example to illustrate:

```
BankAccount account1 = new BankAccount("Savings", 100.0);
BankAccount account2 = new BankAccount("Savings", 100.0);
BankAccount account3 = account1;
```

```
account1 == account2
```

Returns `false` - different objects

```
account1 == account3
```

Returns `true` - same object (aliasing)

Q:

What does the following expressions evaluate to?

```java
int x = 5;
int y = 5;
BankAccount a1 = new BankAccount("Checking");
BankAccount a2 = new BankAccount("Checking");
BankAccount a3 = a1;
int[] arr1 = {1, 2, 3};
int[] arr2 = {1, 2, 3};
int[] arr3 = arr1;
```

- `x == y`
- `a1 == a2`
- `a1 == a3`
- `arr1 == arr2`
- `arr1 == arr3`
- `arr1[0] == arr2[0]`

Enums

What is an Enum?

An **enumeration** (or **enum**) is a special data type that represents a fixed set of constants.

Enums restrict a variable to a predefined set of values.

Examples:
- Days of the week (Monday, Tuesday, ...)
- Card suits (Hearts, Diamonds, Clubs, Spades)
- Directions (North, South, East, West)
- Traffic light states (Red, Yellow, Green)

Enums increase **readability** and **type-safety** by preventing invalid values.

Example: TrafficLight

We can define an enum for traffic light states:

```java
public enum TrafficLight {
    RED, YELLOW, GREEN
}
```

We can use the enum as follows:

```java
public static void getAction(TrafficLight light) {
    if (light == TrafficLight.RED) {
        System.out.println("Stop!");
    } else if (light == TrafficLight.GREEN) {
        System.out.println("Go!");
    }
}
```

Example: DayOfWeek

We can define an enum for days of the week:

```java
public enum DayOfWeek {
    MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY, SUNDAY
}
```

We can use the enum as follows:

```java
public static boolean isWeekend(DayOfWeek day) {
    return day == DayOfWeek.SATURDAY || day == DayOfWeek.SUNDAY;
}
```

Non-Example: Planets

We can define an enum for the planets in our solar system:

```java
public enum Planet {
    MERCURY, VENUS, EARTH, MARS, JUPITER, SATURN, URANUS, NEPTUNE
}
```

Problem: What about Pluto? It used to be a planet, but is now classified as a dwarf planet.

Q:

When would you use an enum and when a class?

- Coffee sizes
- Email messages
- Game difficulty
- Hotel booking
- Flight seats

Enum Methods

Every Java enum comes with a collection of built-in methods:

Method	**Description**
`String name()`	Returns the name of the enum constant
`int ordinal()`	Returns the position of the constant (starting from 0)
`Type[] values()`	Returns an array of all enum constants
`Type valueOf(String)`	Converts a string to the corresponding enum constant

How many values does the following enum have?

Q:

```
enum Season {
    SPRING, SUMMER, FALL, WINTER
}
```

Garbage Collection

Garbage Collection

In other languages (C, C++):

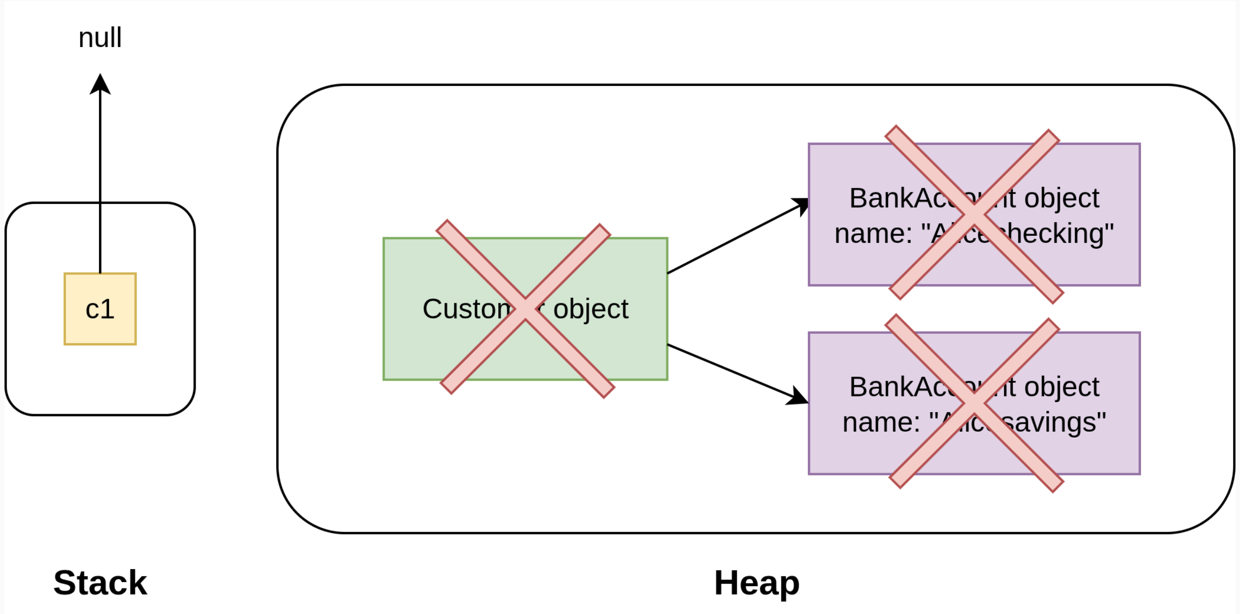
- Programmers manually allocate and deallocate memory
- Must remember to free memory when done
- Easy to make mistakes (memory leaks, crashes)

In Java:

- **Automatic garbage collection**
- Java tracks when objects are no longer referenced
- Memory is automatically freed

Example: Garbage Collection

```java
Customer c = new Customer("Alice");
c = null;
```



Code Style

What is Good Code Style and Why Does it Matter?

Good code style makes programs **easier to read**, **understand**, and **maintain**.

Why does it matter?

- Code is read much more often than it is written.
- Other programmers (including future you!) need to understand your code.
- Consistent style reduces bugs and improves collaboration.

Key principles:

- Use descriptive names and follow naming conventions.
- Organize code consistently.
- Keep methods short and focused.

The **Broken Windows Theory**:
A building with one broken window will soon have *all* its windows broken.

When applied to software: **low quality code** encourages more low quality code.

Example: Good vs Bad Naming

Bad naming:

```
class bank {
    int number;
    double b;

    int get() {
        return number;
    }

    void send(double a) {
        b += a;
    }
}
```

Good naming:

```
class BankAccount {
    private int accountNumber;
    private double balance;

    int getAccountNumber() {
        return accountNumber;
    }

    void deposit(double amount) {
        balance += amount;
    }
}
```

Access Modifiers Best Practices

Key Principles:

- All state (i.e. fields) should be `private`.
- All access must go through getters and setters.
- Do not expose internal details.
- Do not offer setters for fields that do not change.
- All setters must validate their data.

Code Organization

Organize class members in this order:

1. Fields (instance variables)
2. Constructors
3. Methods (getters, setters, then other methods)

Keep methods short:

- Each method should do *one thing* well.
- If a method is too long, split it into smaller methods.
- Aim for methods that fit on one screen.

Use whitespace:

- Blank lines between methods.
- Consistent indentation (usually 4 spaces or 1 tab).

Example: Good Code Style

```java
class Student {
    // 1. Fields
    private final String studentId;
    private String name;
    private int grade;

    // 2. Constructor
    public Student(String studentId,
                   String name) {
        this.studentId = studentId;
        this.name = name;
        this.grade = 0;
    }

    // 3. Getters
    public String getStudentId() {
        return this.studentId;
    }
    // ... continued
```

```java
    // ... continued
    public String getName() {
        return this.name;
    }

    public int getGrade() {
        return this.grade;
    }

    // 4. Setters
    public void setGrade(int newGrade) {
        if (newGrade >= 0 && newGrade <= 100) {
            this.grade = newGrade;
        }
    }

    // 5. Other methods
    public boolean isPassing() {
        return this.grade >= 50;
    }
}
```

Example I: Good Documentation

```
/**
 * Perform a binary search of a range of a char array for a key. The range
 * must be sorted (as by the <code>sort(char[], int, int)</code> method) -
 * if it is not, the behaviour of this method is undefined, and may be an
 * infinite loop. If the array contains the key more than once, any one of
 * them may be found. Note: although the specification allows for an infinite
 * loop if the array is unsorted, it will not happen in this implementation.
 *
 * @param a the array to search (must be sorted)
 * @param low the lowest index to search from.
 * @param hi the highest index to search to.
 * @param key the value to search for
 * @return the index at which the key was found, or -n-1 if it was not
 *         found, where n is the index of the first value higher than key or
 *         a.length if there is no such value.
 * @throws IllegalArgumentException if <code>low > hi</code>
 * @throws ArrayIndexOutOfBoundsException if <code>low < 0</code> or
 *                                        <code>hi > a.length</code>.
 */
public static int binarySearch(char[] a, int low, int hi, char key) { ... }
```

Example II: Good Documentation

```
public static boolean equals(int[] a1, int[] a2) {
    // Quick test which saves comparing elements of the same array, and also
    // catches the case that both are null.
    if (a1 == a2) {
        return true;
    }

    if (null == a1 || null == a2) {
        return false;
    }

    // If they're the same length, test each element
    if (a1.length == a2.length) {
        int i = a1.length;
        // ... omitted ...
        return true;
    }
    return false;
}
```

Bad Code Style

What is the problem here?

```
public class RollLoadedDie {
    public static void main(String[] args) {

        int result = 0;

            double roll = Math.random();
            if( roll <= 0.125 ) result = 1;
                else if( roll <= 0.25) result = 2;
                     else if( roll <= 0.375) result = 3;
                          else if( roll <= 0.5) result = 4;
                              else if( roll <= 0.625) result = 5;
                                    else result = 6;
        System.out.println(result);
    }
}
```

What is the problem here?

```java
            if (isBlack<Math.random())
            {
                StdDraw.setPenColor(Color.black);
                double randomRadius = minRadius + (Math.random() * (maxRadius - minRadius));
                StdDraw.filledCircle(Math.random(),Math.random(), randomRadius);
            }
            else
            {
                StdDraw.setPenColor(Color.white);
                double randomRadius = minRadius + (Math.random() * (maxRadius - minRadius));
                StdDraw.filledCircle(Math.random(),Math.random(), randomRadius);
            }
        }
    }
```

What is the problem here?

```
public class WindChill {
    public static void main(String[] args) {

        /* more code here */

        if ((Math.abs(temperature)>50) || (velocity > 120) || (velocity < 3)){
            System.out.println("values are out of bounds - check your velocity and temparature");
        }
            else
            {   System.out.println("the Wind Chill is " + windChill);
            }
        }
    }
```

What is the problem here?

```
        for (int i = 0; i < n; i++) {
            for (int j = i + 1; j < n; j++){
                if (Math.random() < p){
                    StdDraw.line(x[i], y[i], x[j], y[j]);
                }
            }
        }




    }
}
```

Code Style

What is the problem here?

```
                    }

              }






                          }
        }
```

What is the problem here?

```
            }

        }




                }
    }
```

The T.A. described this as **indentation madness**.

Epilogue

Error of the Week (1/3)

What is the problem here?

```java
public class Student {
    private String firstName;
    private String lastName;

    public Student(String firstName) {
        this.firstName = firstName;
    }

    int nameLength() {
        return firstName.length() +
lastName.length();
    }

    public static void main(String[] args) {
        Student s = new Student("Alice");
        int x = s.nameLength();
    }
}
```

Error of the Week (1/3)

What is the problem here?

```java
public class Student {
    private String firstName;
    private String lastName;

    public Student(String firstName) {
        this.firstName = firstName;
    }

    int nameLength() {
        return firstName.length() +
lastName.length();
    }

    public static void main(String[] args) {
        Student s = new Student("Alice");
        int x = s.nameLength();
    }
}
```

```
Exception in thread "main"
java.lang.NullPointerException: Cannot
invoke "String.length()" because
"this.lastName" is null
  at Student.nameLength(Student.java:11)
  at Student.main(Student.java:16)
```

Error of the Week (2/3)

What is the problem here?

```java
public class Student {
    private final String firstName;
    private final String lastName;

    public Student(String firstName, String lastName) {
        this.firstName = firstName;
        this.lastName = lastName;
    }

    public void setFirstName(String firstName) {
        this.firstName = firstName;
    }

    public static void main(String[] args) {
        Student s = new Student("Alice", "Ross");
    }
}
```

Error of the Week (2/3)

What is the problem here?

```java
public class Student {
    private final String firstName;
    private final String lastName;

    public Student(String firstName, String lastName) {
        this.firstName = firstName;
        this.lastName = lastName;
    }

    public void setFirstName(String firstName) {
        this.firstName = firstName;
    }

    public static void main(String[] args) {
        Student s = new Student("Alice", "Ross");
    }
}
```

```
java: cannot assign a value to
final variable firstName
```

What is the problem here?

```java
public class Teacher {
    Student[] students;

    int totalGrade() {
        int total = 0;
        for (int i = 0; i < students.length; i++) {
            total += students[i].grade;
        }
        return total;
    }
}
```

```java
public class Student {
    private final String name;
    private int grade = 0;

    public Student(String name) {
        this.name = name;
    }

    public int getGrade() {
        return this.grade;
    }
}
```

Error of the Week (3/3)

What is the problem here?

```java
public class Teacher {
    Student[] students;

    int totalGrade() {
        int total = 0;
        for (int i = 0; i < students.length; i++) {
            total += students[i].grade;
        }
        return total;
    }
}
```

```java
public class Student {
    private final String name;
    private int grade = 0;

    public Student(String name) {
        this.name = name;
    }

    public int getGrade() {
        return this.grade;
    }
}
```

```
java: grade has private access in Student
```

Live Programming

Live Programming

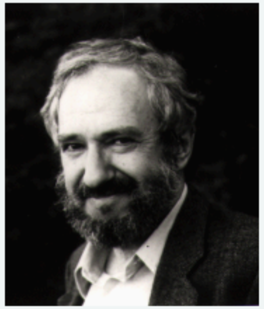
- BankAccount and Customer
  - with joint account
- Null

Turtle Graphics

ADT for turtle graphics

A turtle is an idealized model of a plotting device.

An ADT allows us to write Java programs that manipulate turtles.



Seymour Papert
1928–

Values

position (x, y)	(.5, .5)	(.25, .75)	(.22, .12)
orientation	90°	135°	10°

API (operations)

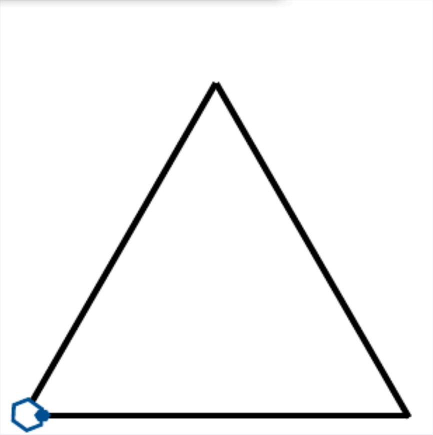
`public class Turtle`	
`Turtle(double x0, double y0, double q0)`	
`void turnLeft(double delta)`	*rotate* `delta` *degrees counterclockwise*
`void goForward(double step)`	*move distance* `step`*, drawing a line*

Turtle graphics implementation: Test client

Best practice. Begin by implementing a simple test client.


```
public static void main(String[] args)
{
   Turtle turtle = new Turtle(0.0, 0.0, 0.0);
   turtle.goForward(1.0);
   turtle.turnLeft(120.0);
   turtle.goForward(1.0);
   turtle.turnLeft(120.0);
   turtle.goForward(1.0);
   turtle.turnLeft(120.0);
}
```

```
% java Turtle
```



Note: Client drew triangle
without computing $\sqrt{3}$

What we *expect*, once the implementation is done.

Turtle implementation: Instance variables and constructor

Instance variables define data-type values.

Constructors create and initialize new objects.

```
public class Turtle
{
    private double x, y;
    private double angle;

    public Turtle(double x0, double y0, double a0)
    {
       x = x0;
       y = y0;
       angle = a0;
    }
...
}
```

instance variables are *not* final

instance variables

constructor

methods

test client

Values

position (*x*, *y*)	(.5, .5)	(.75, .75)	(.22, .12)
orientation	90°	135°	10°
			

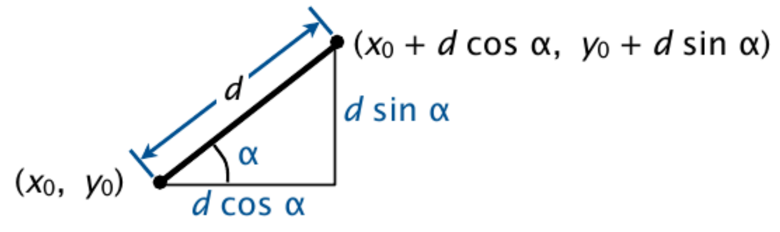
Turtle implementation: Methods

Methods define data-type operations (implement APIs).


```
public class Turtle
{
 ...
   public void turnLeft(double delta)
   { angle += delta; }
   public void goForward(double d)
   {
      double oldx = x;
      double oldy = y;
      x += d * Math.cos(Math.toRadians(angle));
      y += d * Math.sin(Math.toRadians(angle));
      StdDraw.line(oldx, oldy, x, y);
   }
 ...
}
```

API

public class Turtle	
Turtle(double x0, double y0, double q0)	
void turnLeft(double delta)	
void goForward(double step)	



Turtle implementation

text file named `Turtle.java`

```
public class Turtle
{
   private double x, y;
   private double angle;

   public Turtle(double x0, double y0, double a0)
   {
      x = x0;
      y = y0;
      angle = a0;
   }

   public void turnLeft(double delta)
   { angle += delta; }
   public void goForward(double d)
   {
      double oldx = x;
      double oldy = y;
      x += d * Math.cos(Math.toRadians(angle));
      y += d * Math.sin(Math.toRadians(angle));
      StdDraw.line(oldx, oldy, x, y);
   }

   public static void main(String[] args)
   {
     Turtle turtle = new Turtle(0.0, 0.0, 0.0);
     turtle.goForward(1.0); turtle.turnLeft(120.0);
     turtle.goForward(1.0); turtle.turnLeft(120.0);
     turtle.goForward(1.0); turtle.turnLeft(120.0);
   }
}
```

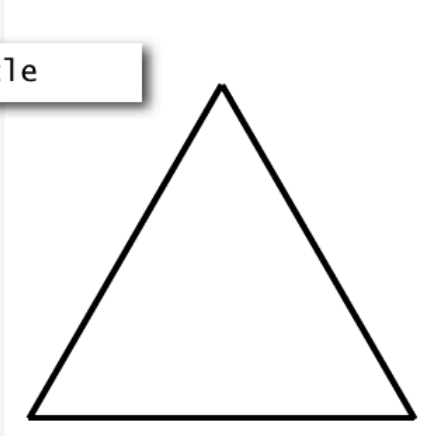
instance variables

constructor

methods

test client

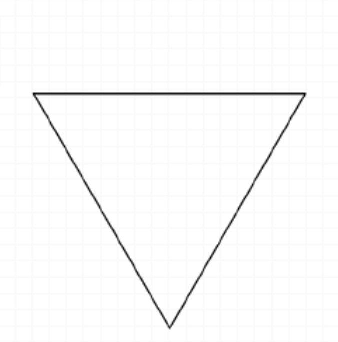
```
% java Turtle
```



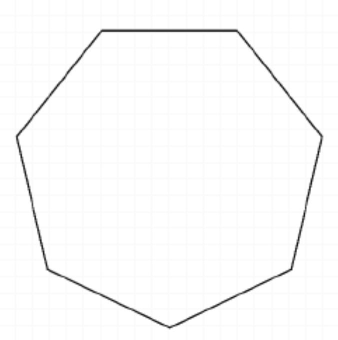
Turtle client: N-gon

```
public class Ngon
{
   public static void main(String[] args)
   {
      int N        = Integer.parseInt(args[0]);
      double angle = 360.0 / N;
      double step  = Math.sin(Math.toRadians(angle/2.0));
      Turtle turtle = new Turtle(0.5, 0, angle/2.0);
      for (int i = 0; i < N; i++)
      {
         turtle.goForward(step);
         turtle.turnLeft(angle);
      }
   }
}
```

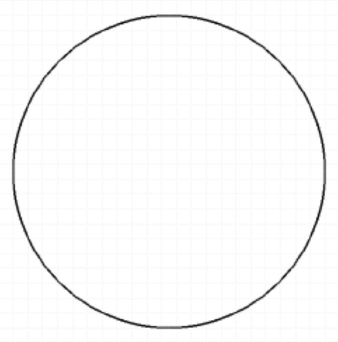
```
% java Ngon 3
```



```
% java Ngon 7
```



```
% java Ngon 1440
```

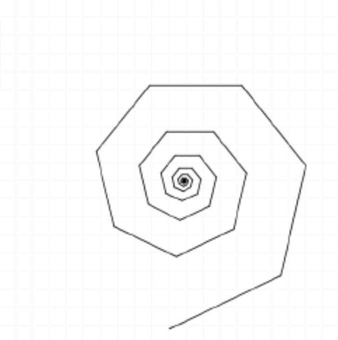


Turtle client: Spira Mirabilis

```
public class Spiral
{
   public static void main(String[] args)
   {
      int N        = Integer.parseInt(args[0]);
      double decay = Double.parseDouble(args[1]);
      double angle = 360.0 / N;
      double step  = Math.sin(Math.toRadians(angle/2.0));
      Turtle turtle = new Turtle(0.5, 0, angle/2.0);
      for (int i = 0; i < 10 * N; i++)
      {
         step /= decay;
         turtle.goForward(step);
         turtle.turnLeft(angle);
      }
   }
}
```

```
% java Spiral 3 1.2
```

```
% java Spiral 7 1.2
```



```
% java Spiral 1440 1.0004
```